Verification and Refactoring

Better Scientific Software Tutorial

Anshu Dubey
Mathematics and Computer Science Division
Argonne National Laboratory

Supercomputing 2018
Dallas, TX
November 12, 2018

License, citation, and acknowledgements

License and Citation

- Requested citation: Anshu Dubey, Verification and Refactoring, Better Scientific Software tutorial, in SC '18: International Conference for High Performance Computing, Networking, Storage and Analysis, Dallas, Texas, 2018. DOI: 10.6084/m9.figshare.7303946.

Acknowledgements

- **Alicia Klinvex contributed many slides to this presentation**
- This work was supported by the U.S. Department of Energy Office of Science, Office of Advanced Scientific Computing Research (ASCR), and by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.
- This work was performed in part at the Argonne National Laboratory, which is managed by UChicago Argonne, LLC for the U.S. Department of Energy under Contract No. DE-AC02-06CH11357

Verification

Definitions

Why is testing important?

Verification

- Code verification uses tests
  - It is much more than a collection of tests
- It is the holistic process through which you ensure that
  - Your implementation shows expected behavior,
  - Your implementation is consistent with your model,
  - Science you are trying to do with the code can be done.

Stages and types of verification

- During initial code development
  - Accuracy and stability
  - Matching the algorithm to the model
  - Interoperability of algorithms
- In later stages
  - While adding new major capabilities or modifying existing capabilities
  - Ongoing maintenance
  - Preparing for production

Stages and types of verification

- If refactoring
  - Ensuring that behavior remains consistent and expected
- All stages have a mix of automation and human-intervention

Note that the stages apply to the whole code as well as its components

Other specific verification challenges

- Functionality coverage
- Particularly true of codes that allow composability in their configuration
- Codes may incorporate some legacy components
  - Its own set of challenges
    - No existing tests at any granularity
- Examples – multiphysics application codes that support multiple domains

Benefits of testing

- Promotes high-quality software that delivers correct results and improves confidence
- Increases quality and speed of development, reducing development and maintenance costs
- Maintains portability to a variety of systems and compilers
- Helps in refactoring
  - Avoid introducing new errors when adding new features
  - Avoid reintroducing old errors

How common are bugs?

Programs do not acquire bugs as people acquire germs, by hanging around other buggy programs. Programmers must insert them.
- Harlan Mills

- Bugs per 1000 lines of code (KLOC)
- Industry average for delivered software
  - 1-25 errors
- Microsoft Applications Division
  - 10-20 defects during in-house testing
  - 0.5 in released product

Code Complete (Steven McConnell)

Why testing is important:
the protein structures of Geoffrey Chang

- Some inherited code flipped two columns of data, inverting an electron-density map
- Resulted in an incorrect protein structure
- Retracted 5 publications
  - One was cited 364 times
- Many papers and grant applications conflicting with his results were rejected

He found and reported the error himself

Why testing is important:
the 40 second flight of the Ariane 5

- Ariane 5: a European orbital launch vehicle meant to lift 20 tons into low Earth orbit
- Initial rocket went off course, started to disintegrate, then self-destructed less than a minute after launch
- Seven variables were at risk of leading to an Operand Error (due to conversion of floating point to integer)
  - Four were protected
- Investigation concluded insufficient test coverage as one of the causes for this accident
- Resulted in a loss of $370,000,000.

Why testing is important:
the Therac-25 accidents

- Therac-25: a computer-controlled radiation therapy machine
- Minimal software testing
- Race condition in the code went undetected
- Unlucky patients were struck with approximately 100 times the intended dose of radiation, ~ 15,000 rads
- Error code indicated that no dose of radiation was given, so operator instructed machine to proceed
- Recalled after six accidents resulting in death and serious injuries

Test Definitions

- Unit tests
  - Test individual functions or classes
- Integration tests
  - Test interaction, build complex hierarchy
- System level tests
  - At the user interaction level

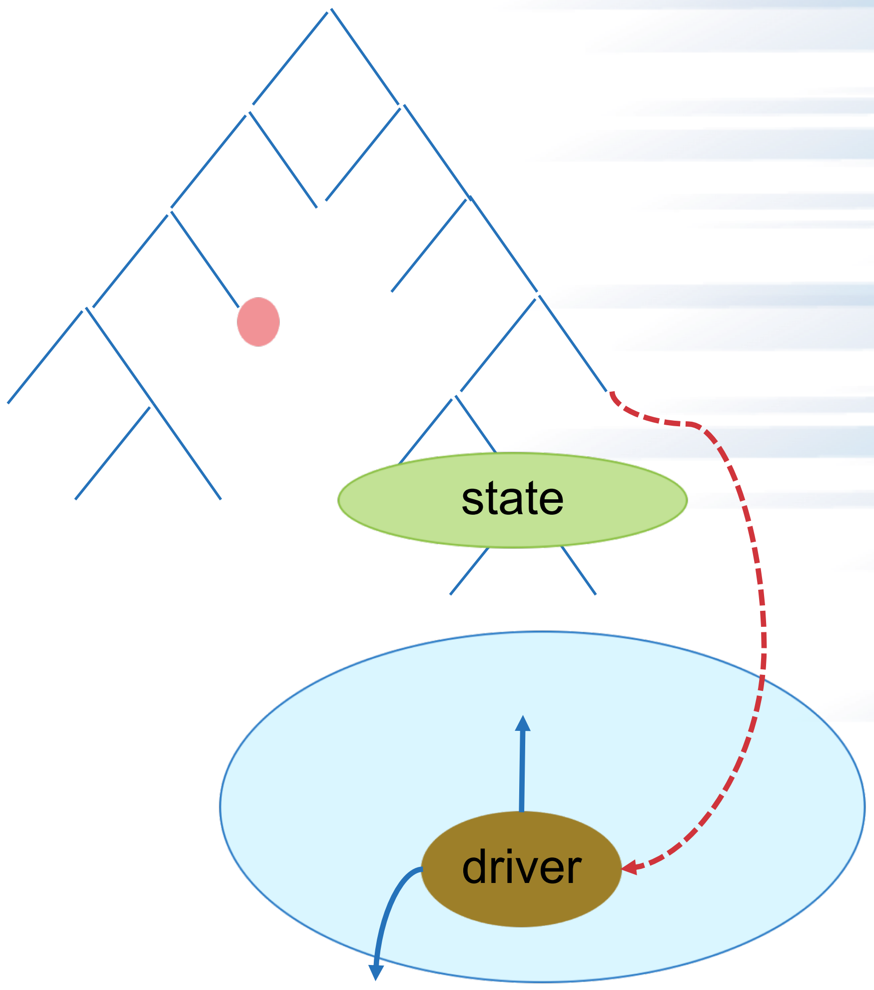
- Restart tests
  - Code starts transparently from a checkpoint
- Regression (no-change) tests
  - Compare current observable output to a gold standard
- Performance tests
  - Focus on the runtime and resource utilization

Test Development

- Development of tests and diagnostics goes hand-in-hand with code development
  - Non-trivial to devise good tests, but extremely important
  - Compare against simpler analytical or semi-analytical solutions
- When faced with legacy codes with no existing tests
  - More creative approach becomes necessary
- Verify correctness
  - Always inject errors to verify that the test is working

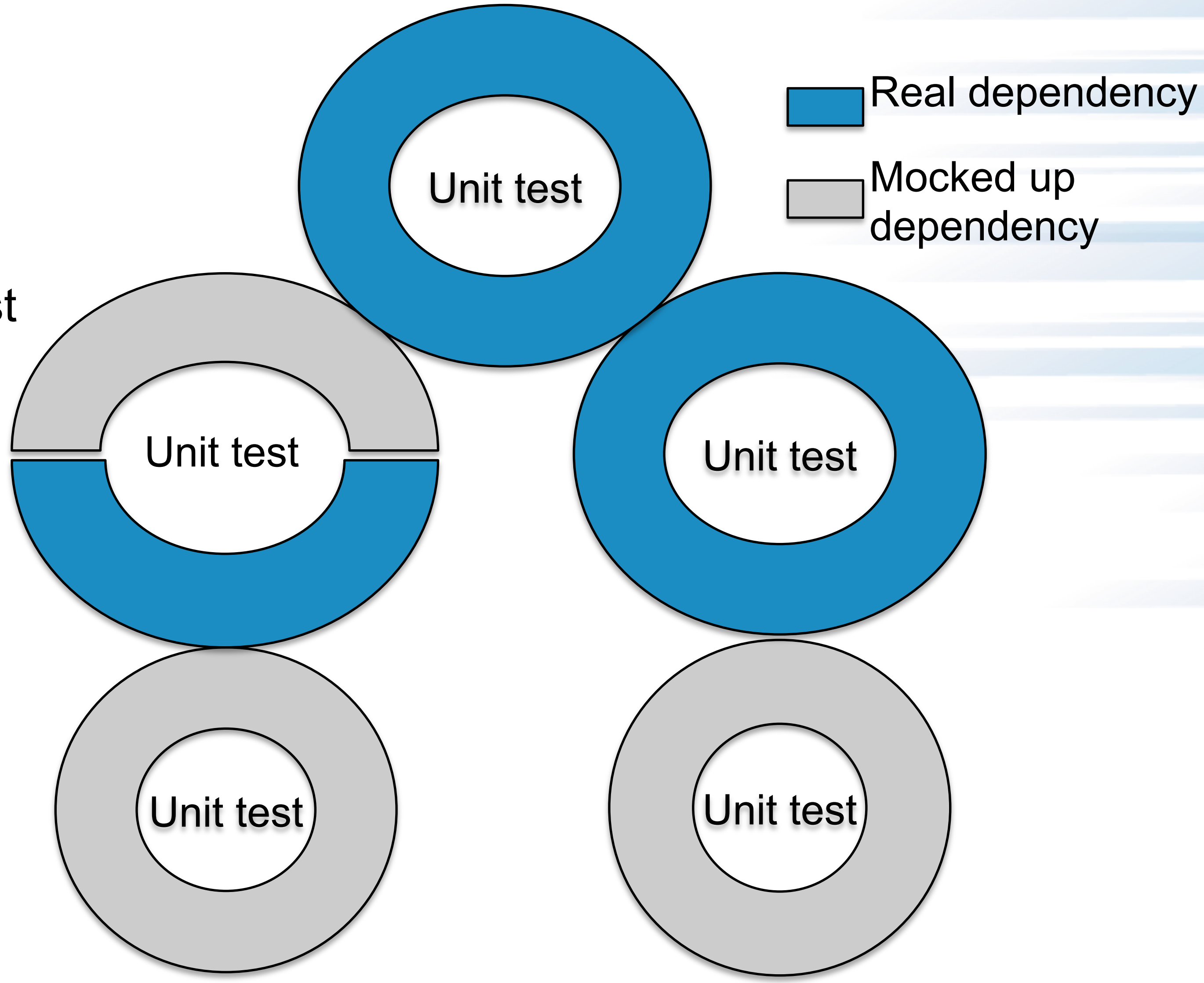
Example from E3SM

- Isolate a small area of the code
- Dump a useful state snapshot
- Build a test driver
  - Start with only the files in the area
  - Link in dependencies
    - Copy if any customizations needed
- Read in the state snapshot
- Restart from the saved state



Workarounds for Granularity

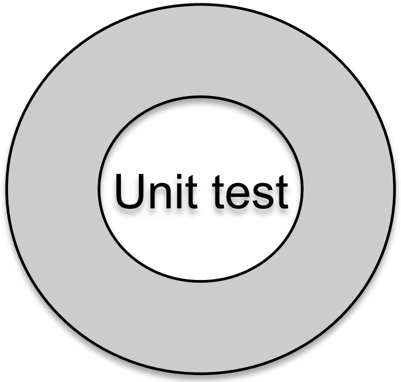
- Approach the problem sideways
  - Components can be exercised against known simpler applications
  - Same applies to combination of components
- Build a scaffolding of verification tests to gain confidence

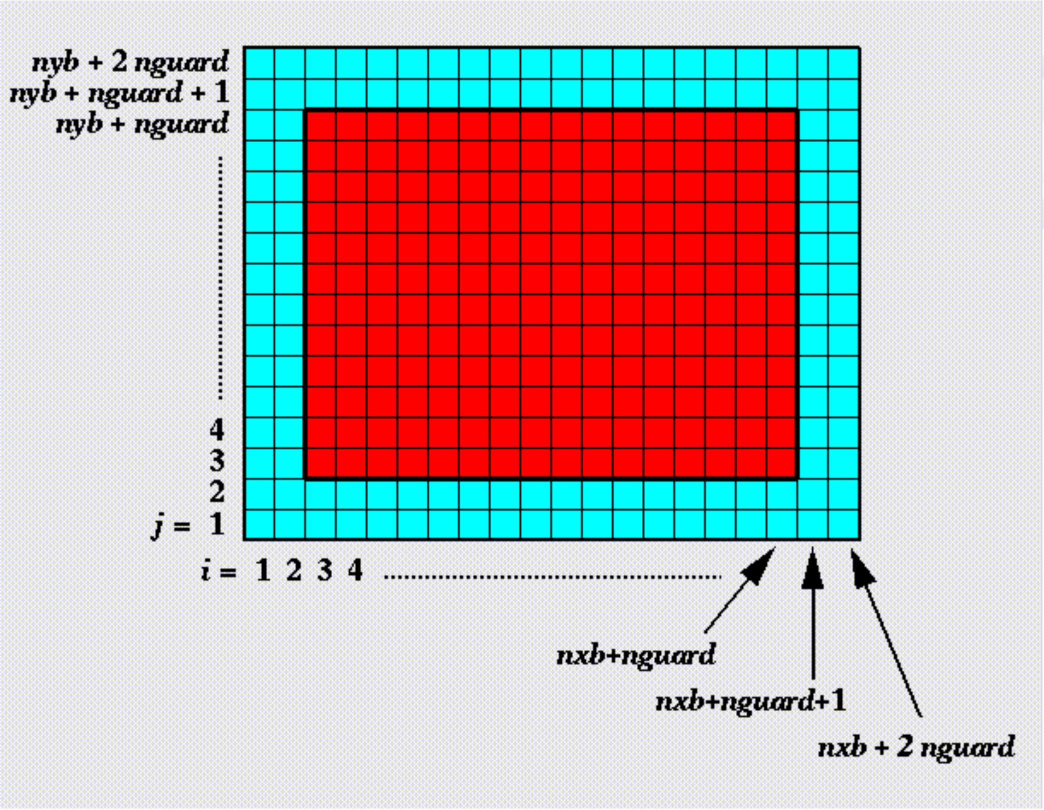


Example from FLASH

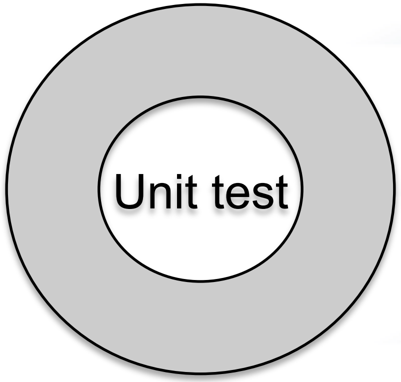
Unit test for Grid

- Verification of guard cell fill
- Use two variables A & B
- Initialize A in all cells and B only in the interior cells (red)
- Apply guard cell fill to B





Example from Flash



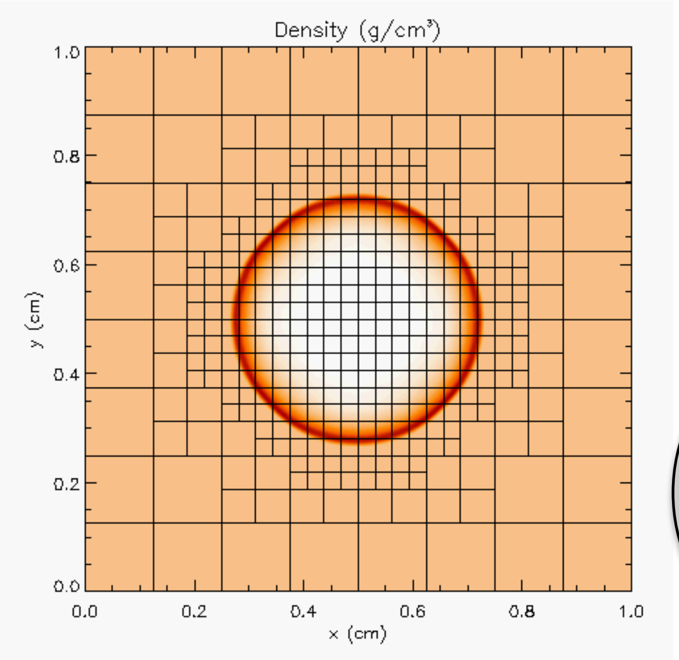
Unit test for Equation of State (EOS)

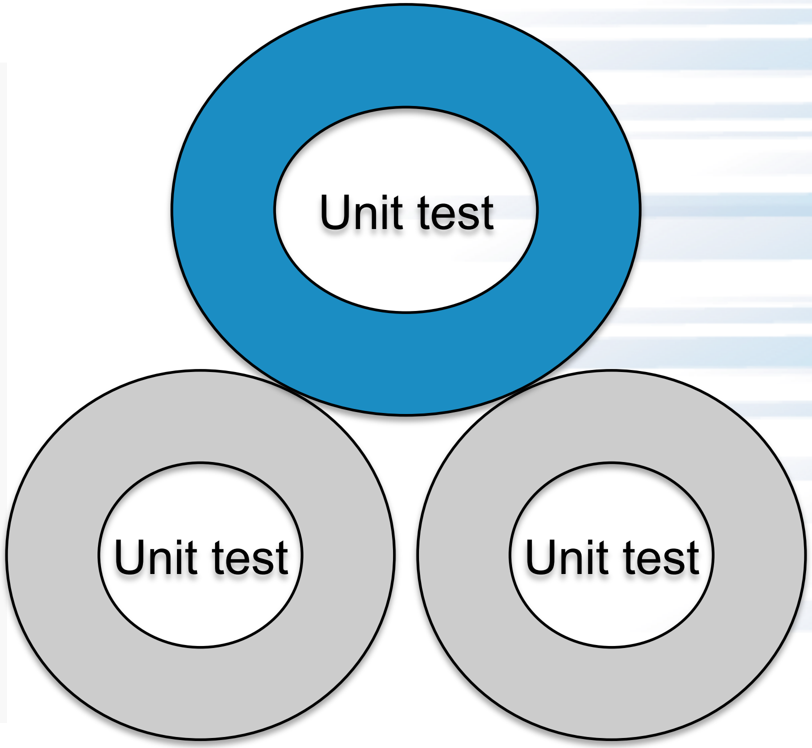
- Three modes for invoking EOS
  - MODE1: Pressure and density as input, internal energy and temperature as output
  - MODE2: Internal energy and density as input temperature and pressure as output
  - MODE3: Temperature and density as input pressure and internal energy as output
- Use initial conditions from a known problem, initialize pressure and density
- Apply EOS in MODE1
- Using internal energy generated in the previous step apply EOS in MODE2
- Using temperature generated in the previous step apply EOS in MODE3
- At the end all variables should be consistent within tolerance

Example from FLASH

Unit test for Hydrodynamics

- Sedov blast wave
- High pressure at the center
- Shock moves out spherically
- FLASH with AMR and hydro
- Known analytical solution





Though it exercises mesh, hydro and eos, if mesh and eos are verified first, then this test verifies hydro

More testing needed for Grid using AMR
Flux correction and regridding

Example from FLASH

Reason about correctness for testing Flux correction and regridding

IF Guardcell fill and EOS unit tests passed

- Run Hydro without AMR
  - If failed fault is in Hydro
- Run Hydro with AMR, but no dynamic refinement
  - If failed fault is in flux correction
- Run Hydro with AMR and dynamic refinement
  - If failed fault is in regridding

Selection of tests

- Two purposes
  - Regression testing
    - May be long running
    - Provide comprehensive coverage
  - Continuous integration
    - Quick diagnosis of error
- A mix of different granularities works well
  - Unit tests for isolating component or sub-component level faults
  - Integration tests with simple to complex configuration and system level
  - Restart tests
- Rules of thumb
  - Simple
  - Enable quick pin-pointing

Why not always use the most stringent testing?

- Effort spent in devising tests and testing regime are a tax on team resources
- When the tax is too high…
  - Team cannot meet code-use objectives
- When is the tax is too low…
  - Necessary oversight not provided
  - Defects in code sneak through
- Evaluate project needs
  - Objectives: expected use of the code
  - Team: size and degree of heterogeneity
  - Lifecycle stage: new or production or refactoring
  - Lifetime: one off or ongoing production
  - Complexity: modules and their interactions

Commonalities

- Unit testing is always good
  - It is never sufficient
- Verification of expected behavior
- Understanding the range of validity and applicability is always important
  - Especially for individual solvers

Test Selection

- First line of defense – code coverage tools (demo later)
- Necessary but not sufficient – don't give any information about interoperability

- Build a matrix
  - Physics along rows
  - Infrastructure along columns
  - Alternative implementations, dimensions, geometry
- Mark <i,j> if test covers corresponding features
- Follow the order
  - All unit tests – including full module tests
  - Tests representing ongoing productions
  - Tests sensitive to perturbations
  - Most stringent tests for solvers
  - Least complex test to cover remaining spots

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Example

	Hydro	EOS	Gravity	Burn	Particles
AMR	CL	CL		CL	CL
UG	SV	SV			SV
Multigrid	WD	WD	WD	WD	
FFT			PT		

Tests	Symbol
Sedov	SV
Cellular	CL
Poisson	PT
White Dwarf	WD

- A test on the same row indicates interoperability between corresponding physics
- Similar logic would apply to tests on the same column for infrastructure
- More goes on, but this is the primary methodology

IDEAS productivity

ECP EXASCALE COMPUTING PROJECT

Regular Testing

- Part of ongoing verification
- Automating is helpful
- Can be just a script
- Or a testing harness

Jenkins
C-dash
Custom
(FlashTest)

- Essential for large code
  - Set up and run tests
  - Evaluate test results
- Easy to execute a logical subset of tests
  - Pre-push
  - Nightly
- Automation of test harness is critical for
  - Long-running test suites
  - Projects that support many platforms

Regular Testing

- Testing regime is only useful if it is
  - Maintained
    - Tests and benchmarks periodically updated
  - Monitored regularly
    - Can be automated
  - Has rapid response to failure
    - Tests should pass most of the time

Refactoring

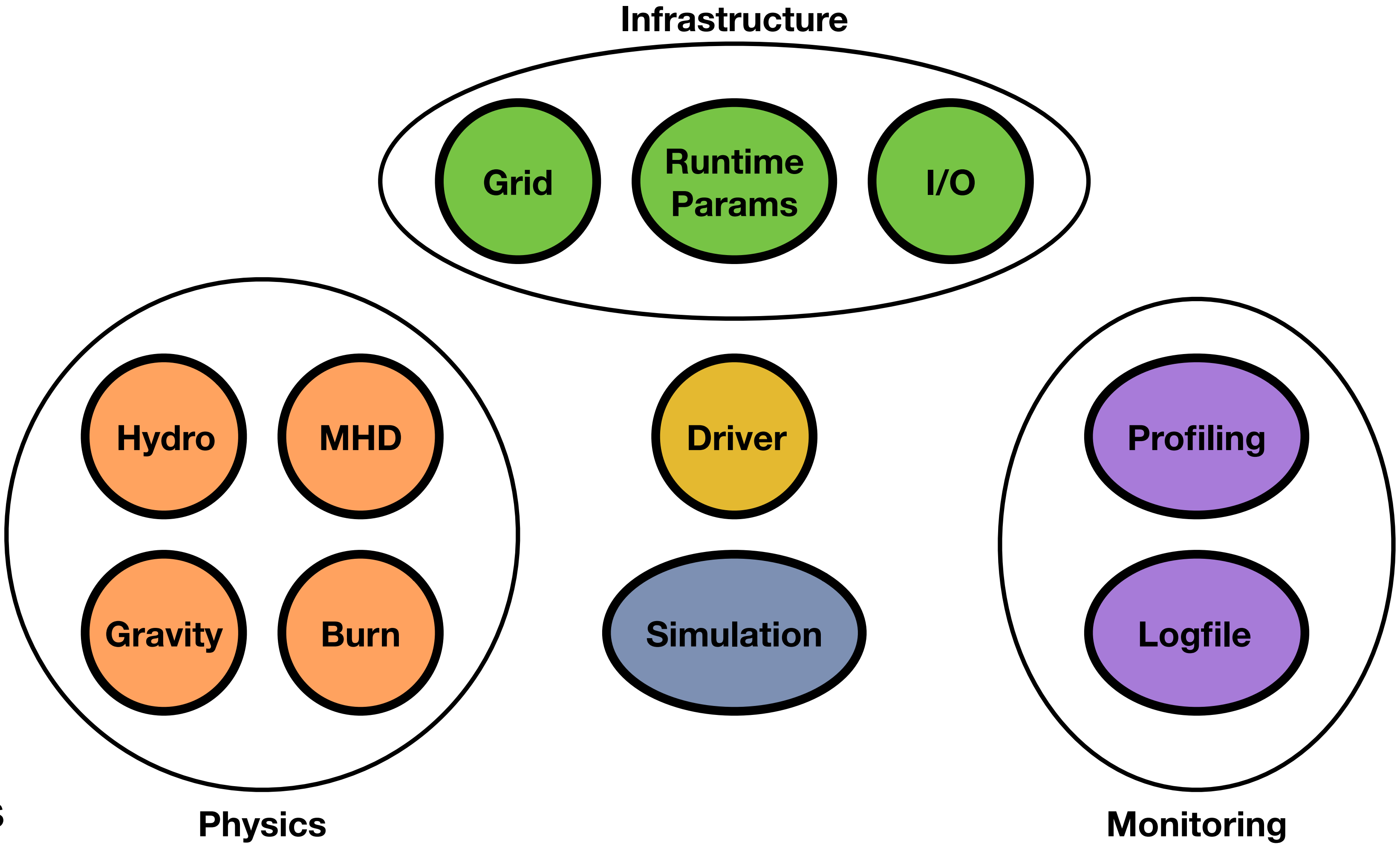
Considerations

- Know bounds on acceptable behavior change
- Know your error bounds
  - Bitwise reproduction of results unlikely after transition
- Map from here to there
- Check for coverage provided by existing tests
- Develop new tests where there are gaps

Incorporate testing overheads into refactor cost estimates

Example FLASH

- Grid
  - Manages data
  - Domain discretization
- Hydro
  - simpleUnsplit
  - Unsplit
- Driver
  - Time-stepping
  - Orchestrates interactions



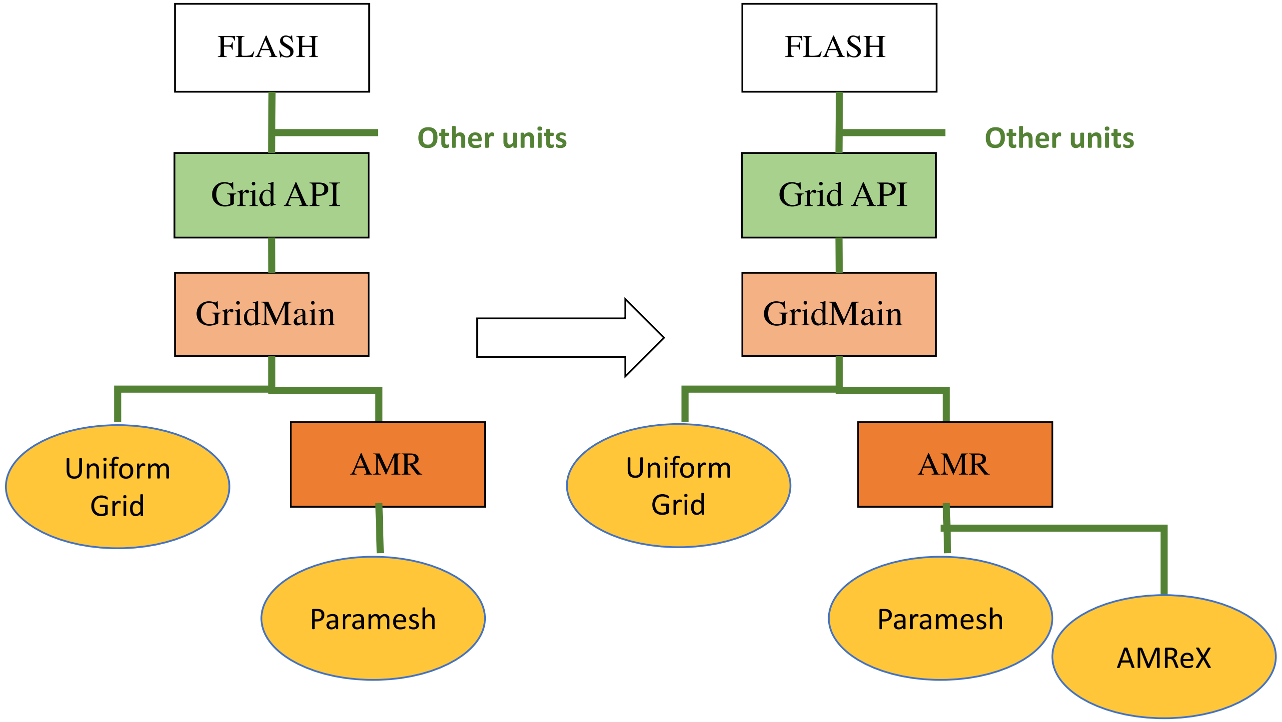
FLASH5

Refactoring for Next Generation Hardware

AMReX - Lawrence Berkeley National Lab
- Designed for exascale
- Node-level heterogeneity
- Smart iterators hide parallelization

Goal: Replace Paramesh with AMReX

Plan:
- Paramesh & AMReX coexist
- Adapt interfaces to suit AMReX
- Refactor Paramesh implementation
- Compare AMReX implementation against Paramesh implementation



Refactoring plan

Design

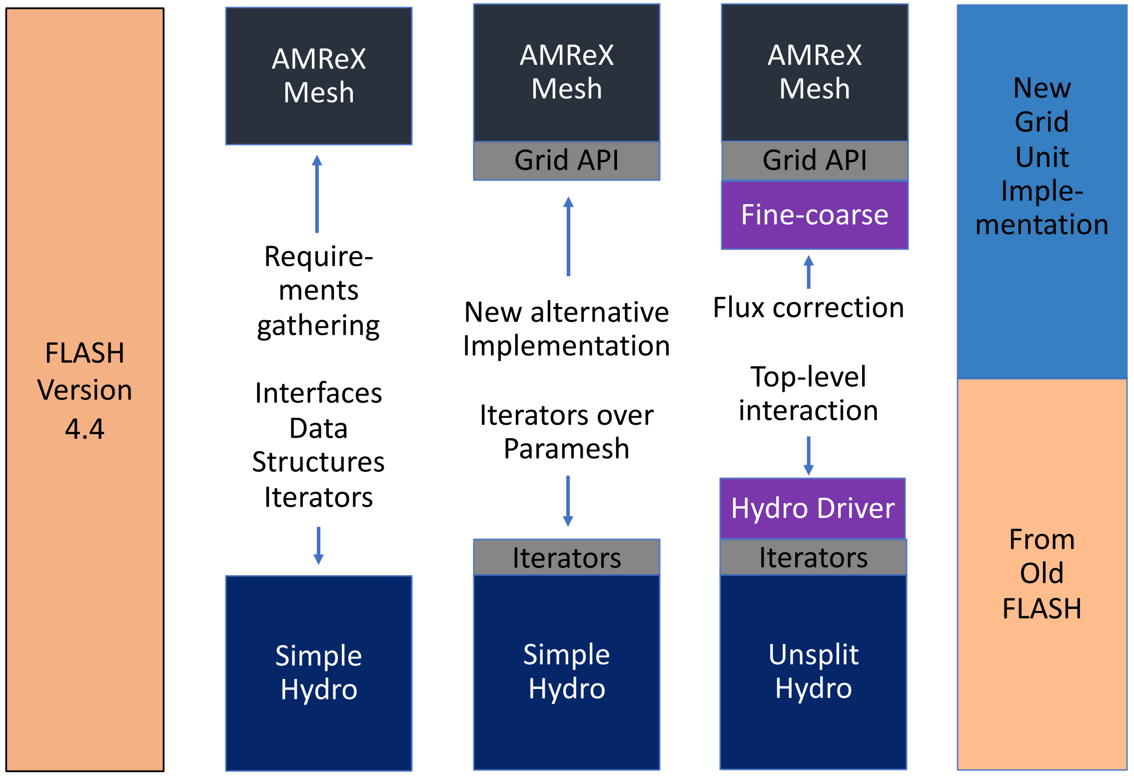
- Degree & scope of change
- Formulate initial requirements

Prototyping

- Explore & test design decisions
- Update requirements

Implementation

- Recover from prototyping
- Expand & implement design decisions



Phase 1 - design

Sit, think, hypothesize, & argue

- Derive and understand principal definitions & abstractions
- Collect & understand Paramesh/AMReX constraints
  - Generally useful design due to two sets of constraints?
- Collect & understand physics unit requirements on Grid unit
- Design fundamental data structures & update interface
  - AMReX introduces iterators over blocks/tiles of mesh
  - Package up block/tile index with associated mesh metadata
- Minimal prototyping with no verification

Phase 2 - prototyping

Quick, dirty, & light

- Implement new data structures
  - Evolve design/implementation by iterating between Paramesh & AMReX
- Explore Grid/physics unit interface
  - `simpleUnsplit` Hydro unit
- Discover use patterns of data structures and Grid unit interface
- Adjust requirements & interfaces

Verification

- Single `simpleUnsplit` simulation
- Quantitative regression test with Paramesh
- Proof of concept with AMReX *via* qualitative comparison with Paramesh

Phase 3 - implementation

Toward quantifiable success & Continuous Integration

- Derive & implement lessons learned
  - Clean code & inline documentation
- Update Unsplit Hydro
- Hybrid FLASH
  - AMReX manages data
  - Paramesh drives AMR
- Fully-functioning simulation with AMReX
- Prune old code

Verification

- Git workflow
- Grow test suite / CI with Jenkins
- Add new feature/test
  - Create Paramesh baseline with FLASH4.4
  - Refactor Paramesh implementation
  - Implement with AMReX & compare against Paramesh baseline

Other resources

Software testing levels and definitions:
http://www.tutorialspoint.com/software_testing/software_testing_levels.htm

Working Effectively with Legacy Code, Michael Feathers. The legacy software change algorithm described in this book is very straight-forward and powerful for anyone working on a code that has insufficient testing.

Code Complete, Steve McConnell. Includes testing advice.

Organization dedicated to software testing: https://www.associationforsoftwaretesting.org/

Software Carpentry: http://katyhuff.github.io/python-testing/

Tutorial from Udacity: https://www.udacity.com/course/software-testing--cs258

Papers on testing:
http://www.sciencedirect.com/science/article/pii/S0950584914001232
https://www.researchgate.net/publication/264697060_Ongoing_verification_of_a_multiphysics_community_code_FLASH

Resources for Trilinos testing:
Trilinos testing policy: https://github.com/trilinos/Trilinos/wiki/Trilinos-Testing-Policy
Trilinos test harness: https://github.com/trilinos/Trilinos/wiki/Policies--%7C-Testing

Agenda

Tutorial evaluation form: http://bit.ly/sc18-eval

Time	Module	Topic	Speaker
8:30am-8:40am	00	Introduction and Setup	David E. Bernholdt, ORNL
8:40am-9:00am	01	Overview of Best Practices in HPC Software Development	David E. Bernholdt, ORNL
9:00am-10:00am	02	Git Workflows	Jared O'Neal, ANL
10:00am-10:30am		*Break*	
10:30am-11:40am	03	Better (Small) Scientific Software Teams	Michael A. Heroux, SNL
11:40am-12:00pm	04	Improving Reproducibility through Better Software Practices	Michael A. Heroux, SNL
12:00pm-1:30pm		*Lunch (C1/2/3/4 Ballroom, 2nd floor)*	
1:30pm-2:15pm	05	An Introduction to Software Licensing	David E. Bernholdt, ORNL
2:15pm-2:55pm	06	Verification and Refactoring	Anshu Dubey, ANL
2:55pm-3:00pm	07	Code Coverage and Continuous Integration	Jared O'Neal, ANL
3:00-3:30pm		*Break*	
3:30pm-3:40pm	07	Code Coverage and Continuous Integration (continued)	Jared O'Neal, ANL
3:40pm-5:00pm	08	Hands-on Activities	Jared O'Neal, ANL, and team